SHIFTLEFT OCULAR

INTRODUCTION

ShiftLeft Ocular offers code auditors the full range of capabilities of ShiftLeft's best-in-class static code analysis[1], ShiftLeft Inspect. Ocular enables code auditors to write custom queries to identify vulnerabilities by leveraging Inspect's raw analysis engines. With Ocular, security checks can account for your organization's unique environment, including custom code, frameworks, open source libraries, commercial SDKs, and external APIs. The custom queries can reduce false positives precisely modeling high level information flows. Further, Ocular custom queries can find vulnerabilities that can be overlooked by traditional code analysis, such as vulnerabilities due to improperly sanitized or completely unsanitized input that flows across dynamic call sites.

In particular, Ocular's fluent query engine provides access to ShiftLeft's platform elements, which include the following:

The Code Property Graph (CPG): A multifaceted semantic code representation

EXAMPLE: In addition to finding vulnerabilities like SQL injection, cross-site scripting, and deserialization, queries to the CPG can also identify code weaknesses, such as methods with too many parameters, improperly sanitized inputs, duplicate code, and inconsistent naming.

The Advanced Static Data-Flow Analyzer: An end-to-end representation of how data flows across all sources and sinks and transforms in the application—including connections across microservices and dependencies, such as open source libraries, commercial SDKs, and external APIs

[1] https://blog.shiftleft.io/beating-the-owasp-benchmark-24a7b1601031)

THE CODE PROPERTY GRAPH

ShiftLeft's core technology is the CPG, which is a fundamentally new and more precise way to rapidly analyze high volumes of source code for vulnerabilities. The CPG leverages semantic graphing to create a single multilayered graph that summarizes code on various levels of abstraction; this includes abstract syntax trees, control flow graphs, call graphs, program dependency graphs, and directory structures. This enables ShiftLeft to understand the context relating to what the application is and is not supposed to do, making it much easier to identify deviations as violations or vulnerabilities. In particular, this is critical for identifying complex vulnerabilities that are dependent on a series of conditions across various components that make up the application. Only by understanding how the components interact with each other can these sophisticated vulnerabilities be easily identified.

EXAMPLE: A custom query against the Advanced Static Data-Flow Analyzer could determine whether a particular flow has sanitization and if sanitization transforms occur in the correct order.

Security Profiles: Summaries of all applications' security-relevant properties, generated with respect to a user-modifiable code-level security policy

EXAMPLE: Queries against Security Profiles can help explore code weaknesses by severity and type.

HOW OCULAR WORKS

Ocular operates on Java archives (JAR or WAR), which are passed to the java2cpg tool, to generate an intermediate graphical representation of code: a CPG. This graph can be queried using an interactive shell, the REPL, or analyzed automatically to generate a security profile with the cpg2sp tool, according to the security policy. The security profile summarizes the security-relevant properties of the application, including identified vulnerabilities and data leaks. Finally, the REPL can be run noninteractively to scan code via custom user scripts.

ADDITIONAL FEATURES

- **Insert Ocular Policies into DevOps Pipelines:** Custom queries can be saved as policies; then, by leveraging the ShiftLeft Platform, they run automatically upon pull request, build, or release.
- **Cross-Language Policies:** Ocular converts programs for each supported programming language to an intermediate representation. This allows the same query to be run across code bases written in multiple programming languages (statically and dynamically typed languages). Hence, Ocular queries can be used to quickly apply and confirm standards across the entire environment, regardless of programming language.
- **Leverage Policy Libraries:** Ocular comes with annotations for common Java frameworks and libraries. Possibly attacker-controlled data sources and interesting sinks are tagged in the graph automatically, and flow descriptions exist to scan for common vulnerability patterns. Users can provide additional annotations to extend supported frameworks and libraries or encode additional vulnerability patterns.
- **Integrate with Security Tools:** Following the UNIX philosophy, Ocular outputs result in standard JSON formats for easy integration with any tools your organization may be using.

DETAILED EXAMPLES

EXAMPLE #1: CUSTOM SANITIZATION

Traditional code analysis tools are often unaware of custom sanitizations that properly secure user input. Hence, unknown or custom sanitization is a source of false positives in traditional code analysis. This example shows how Ocular queries can quickly eliminate sanitization false positives.

The following query filters flows that contain SANITIZATION_METHOD_FULLNAME.

```
1.  sink.reachableBy(source)
2.      .flows.l.filter { flow =>
3.                      !flow.points
4.                          .map(_.location.methodFullName)
5.                          .contains("SANITIZATION_METHOD_FULLNAME")
6.                  }
7.      .toList
8.      .foreach    { flow =>
9.                      println("\nNew Flow\n====")
10.                     println( flow.points
11.                             .map(_.location.methodFullName)
12.                             .mkString("\n")
13.                         )
14.                 }
```

The call to reachableBy in line 1 returns all the flows where the sink is reachable by a source followed by a filtering step. The filtering in lines 2 to 6 iterates over all flows and their entries. As soon as a SANITIZATION_METHOD_FULLNAME is found in a flow, this flow will not be reported. Lines 7 to 14 print the remaining list of potentially vulnerable flows.

EXAMPLE #2: INDIRECT DATA FLOW (GETTER-SETTER)

The task of algorithmically finding indirect data flows in applications can be quite challenging and is a common source of false negatives in traditional code analysis tools. Indirect data flows, that is, flows where user input is not directly used in a sink, are very common in object-oriented languages (such as Java); they often appear in the form of "getter" and "setter" methods. The following simplified example illustrates all the necessary steps needed to find indirect data flows by using ShiftLeft Ocular.

SOURCE CODE

The program below (Test.java) contains a data flow starting from the source, that is, the command line argument (args[0]), through the member name of the User class, ending up in the sink, that is, the FileOutputStream constructor. This program yields a file that is named after the user-provided command line argument (args[0]). The flow is indirect because the user input is first stored in the user object by means of the call to setName (line 15) and is then later retrieved by a call to getName (line 16) before it is used in the FileOutputStream (line 16).

```
1.  import java.io.*;
2.  class User {
3.     private String name;
4.     public void setName(String name) {
5.        this.name = name;
6.     }
7.     public String getName() {
8.        return this.name;
9.     }
10. }
11.
12. public class Test {
13.    public static void main(String[] args) throws Exception {
14.      User u = new User();
15.      u.setName(args[0]);
16.      FileOutputStream fos = new FileOutputStream(u.getName());
17.    }
18. }
```

CREATE THE CPG

Before we can generate a CPG from our source code, we compile and check the behavior of our example code. First, we compile the Test.java program by invoking javac Test.java. Subsequently, we execute the program with java Test testuser with testuser as the command line argument. As we can see in the directory listing, the execution of Test with the command line argument testuser yields a file named testuser.

```
$ javac Test.java
$ java Test testuser
$ ls -alh testuser
-rw-r--r-- 1 ebp users 8 Nov 23 15:21 testuser
```

However, the Java frontend for CPG generation (java2cpg) only accepts .jar files. Thus, we need to pack the generated class file (Test.class) into a .jar file by executing the following command:

```
$ jar cvf test.jar Test.class
```

We can then use the .jar file as a parameter for java2cpg, which will generate a CPG (in the example below test.bin.zip):

```
$ ./java2cpg.sh -f protobufzip -o test.bin.zip test.jar
2018-11-23 15:23:09.486 [main] INFO Using class selection approach: BLACKLIST_WITH_DEFAULT []
2018-11-23 15:23:09.487 [main] INFO Using packaged default blacklist
2018-11-23 15:23:09.494 [main] INFO Applying 1099 blacklist entries to /tmp/java2cpg_
class8356829537693124775.
2018-11-23 15:23:09.858 [Thread-4] INFO Found 33 files
```

After the execution of the above command, you will find the CPG file test.bin.zip in your current working directory.

```
$ ls -alh test.bin.zip
rw------- 1 ebp users 20K Nov 23 15:23 test.bin.zip
```

SEARCHING THE FLOWS

After generating the CPG, we can load it into Ocular and search for the indirect data flow. In the listing below, the command loadCpg (line 1) loads the CPG; the commands in lines 2 and 3 create source and sink variables, respectively; and the call to reachableBy in line 4 triggers the actual reachability search. The data flow engine searches from the arguments of the main method to the parameters of every method containing File OutputStream in its full name.

```
1. ocular> loadCpg("test.bin.zip")
2. ocular> val source = cpg.method.name("main").parameter
3. ocular> val sink = cpg.method.fullName(".*FileOutputStream.*").parameter
4. ocular> sink.reachableBy(source).flows.p
```

The initial search does not deliver any results; this is because we need to first define data flow mappings for capturing indirect data flows. The following mappings can be stored in the file ~/.shiftleft/policy/dynamic/jvm/User.policy:

```
MAP -preserve -d INST -s PAR -i 1 METHOD -f "User.setName:void(java.lang.String)"
MAP -override -d RET -s INST METHOD -f "User.getName:java.lang.String()"
```

The first mapping is responsible for propagating (and preserving) the taint from the first parameter of setName to the User instance, whereas the second mapping is responsible for propagating the taint from the User instance to the return parameter of getName.

After loading the mappings (by loading a CPG), the data flow query should return the following result:

------ Flow with 18 elements ------

```
l0_0    15    main   /Test.java
l0_0[0]       17    main   /Test.java
param1        <operator>.assignment   N/A
param0        <operator>.assignment   N/A
$r0    17    main   /Test.java
$r0    17    main   /Test.java
param0        setName       /User.java
this          setName       /User.java
l1_0    17    main   /Test.java
l1_0    18    main   /Test.java
this          getName       /User.java
$ret          getName       /User.java
l1_0.getName()  18    main   /Test.java
param1        <operator>.assignment   N/A
param0        <operator>.assignment   N/A
$r1    18    main   /Test.java
$r1    18    main   /Test.java
param0        <init>        java/io/FileOutputStream.java
```